Geekbrains

Приближённые вычисления элементарных функций: теоретические основы и практическая реализация

Программа: Разработчик
Специализация: Веб-разработка на Java
Казначеев Матвей Андреевич

Самара 2024

Оглавление

Введение

Элементарные функции, такие как экспонента, логарифм, тригонометрические функции и другие, играют ключевую роль во многих областях науки и техники. Точные значения этих функций часто недоступны или их вычисление требует значительных вычислительных ресурсов. Поэтому приближённые методы вычисления элементарных функций становятся необходимыми для эффективного решения практических задач.

Цель данного проекта — разработать теоретические основы и практические инструменты для приближённых вычислений элементарных функций с контролируемой точностью. В работе рассматриваются численные методы, основанные на разложении функций в ряды Тейлора и другие степенные ряды, что позволяет получить приближённые значения функций с заданной точностью.

В первой части проекта изложены теоретические основы численных методов и вычислительных алгоритмов. Рассматривается структура чисел, методы управления точностью, операции округления, а также способы оценки погрешностей вычислений.

Во второй части описывается реализация и оптимизация вычислительных алгоритмов для приближённых вычислений. Разрабатываются методы для эффективного суммирования рядов, оценки количества необходимых членов ряда для достижения заданной точности, а также алгоритмы для работы в многопоточной среде. Использование объектно-ориентированного подхода и паттернов проектирования позволяет создавать гибкие и расширяемые системы.

Практическая значимость проекта заключается в возможности применения разработанных методов и инструментов в различных областях, требующих точных и эффективных вычислений. Это может быть полезно в финансовых расчётах, инженерных приложениях, научных исследованиях и других сферах, где важна высокая точность и производительность вычислений.

Настоящая работа объединяет теоретические знания и практические навыки программирования, демонстрируя, как современные численные методы могут быть эффективно реализованы с использованием возможностей языка Java и объектно-ориентированного подхода. Проект предоставляет основу для дальнейших исследований и разработок в области численных вычислений и алгоритмов приближения элементарных функций.

1 Теоретические основы численных методов и вычислительных алгоритмов

1.1 Структура чисел и методы управления точностью в вычислительных алгоритмах

1.1.1 Разрядная структура десятичных чисел

Зачастую нам понадобится рассматривать структурные элементы десятичного числа — цифры — по отдельности. Поскольку наши рассуждения редко будут касаться конкретных чисел, но часто будут оперировать числами из конкретных диапазонов, обращаться к той или иной их цифре удобнее посредством указания её позиции. Позицию цифры в записи числа называют **разрядом**. Каждому разряду можно присвоить порядковый номер. Для целой части числа нумерация начинается с нуля и идет справа налево, увеличиваясь на единицу для каждого следующего разряда. Для дробной части нумерация разрядов начинается с минус единицы и идет слева направо, уменьшаясь на единицу для каждого следующего разряда. Разряд с порядковым номером n будем называть n-м разрядом.

В позиционной системе счисления весом n-го разряда называют величину 10^n. Таким образом, значение каждой цифры определяется произведением цифры на вес её разряда. Например, в числе $123{,}45$ каждая цифра умножается на соответствующий вес разряда, что позволяет точно представить значение числа как сумму этих произведений:

$$123{,}45 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2}.$$

Понимание разрядной структуры числа является фундаментальным аспектом разработки алгоритмов вычисления элементарных функций с произвольной точностью.

1.1.2 Разрядная точность чисел в позиционной системе счисления

Под разрядной точностью будем подразумевать характеристику числа, указывающую на количество цифр в его записи. Существует множество подходов к определению разрядной точности. Как мы увидим в дальнейшем, вычислительному алгоритму чаще всего удобно задать точность как номер младшего разряда, который подлежит вычислению. С точки зрения конечного пользователя, удобнее запрашивать вычисления с определённым количеством знаков после запятой или количеством значащих цифр.

Значащими цифрами числа называют все цифры, начиная с первой ненулевой (при чтении числа слева направо) и заканчивая последней. При этом среди последних цифр могут встречаться нули. Говорить о количестве значащих цифр числа ноль бессмысленно.

Обычно точность выражают количеством значащих цифр, когда модуль числа очень маленький или, напротив, очень большой. Такие числа неудобно записывать в обычной десятичной форме из-за большого количества цифр, поэтому прибегают к научной записи. **Научной формой** числа a называют его представление в виде

$$a = \alpha \cdot 10^n, \tag{1.1}$$

где $1 \leqslant |\alpha| < 10$ и $n \in \mathbb{Z}$. Число α называют мантиссой, а число n — экспонентой. Из определения видно, что первая значащая цифра числа вида (1.1) является первой цифрой мантиссы α.

Связь между количеством s цифр после запятой некоторого числа и его младшим разрядом k интуитивно понятна:

$$k = -s. \tag{1.2}$$

При этом, поскольку $s \geqslant 0$, обратное использование этой формулы требует осторожности. Так, $s = -k$ имеет смысл, если $k \leqslant 0$. В противном случае мы получим отрицательное количество цифр после запятой. Поэтому в каждом конкретном случае придётся договариваться о правилах обработки таких ситуаций.

Связь между количеством значащих цифр λ и младшим разрядом k выража-

ется несколько сложнее. Переведём интересующее нас число в научную форму. Очевидно, что его мантисса будет содержать $\lambda - 1$ цифру после запятой, а значит, её младший разряд будет иметь номер $1 - \lambda$. Заметим, что переход к мантиссе осуществляется сдвигом десятичного разделителя исходного числа на n позиций влево, если $n \geqslant 0$, или на $-n$ позиций вправо, если $n < 0$, где n — экспонента числа. То есть младший разряд мантиссы имеет номер $k - n$. Итогом этих размышлений является уравнение

$$1 - \lambda = k - n, \tag{1.3}$$

откуда получим, что

$$k = n + 1 - \lambda. \tag{1.4}$$

Поскольку λ — число строго положительное, нужно снова осторожно применять обратное выражение $\lambda = n + 1 - k$. Рассмотрим гипотетическую ситуацию, когда $k \geqslant n+1$, то есть номер младшего разряда как минимум на единицу превышает экспоненту числа. Это означает, что ни одна значащая цифра числа не входит в диапазон вычисления, поскольку первая из них занимает n-й разряд. В таком случае зависимость между k и λ даст нам отрицательное количество значащих цифр. Эту ситуацию следует обрабатывать как ошибочную.

Понимание полученных соотношений позволяет эффективно управлять точностью при вычислениях и избегать ошибок, связанных с неправильным заданием разрядной точности.

1.1.3 Округление чисел в вычислительных алгоритмах

В дальнейшем нам часто придётся пользоваться функцией округления вещественных чисел. Для любого вещественного числа x **функция округления вверх**, обозначаемая $\lceil x \rceil$, определяется как наименьшее целое число, не меньшее x. То есть:

$$\lceil x \rceil = \min\{n \in \mathbb{Z} \mid n \geqslant x\}.$$

Эквивалентно можно записать: $\lceil x \rceil = c$, где $c \in \mathbb{Z}$ и

$$c - 1 < x \leqslant c.$$

Аналогично, для **округления вниз** имеем

$$\lfloor x \rfloor = \max\{n \in \mathbb{Z} \mid n \leqslant x\},$$

или $\lfloor x \rfloor = f$, где $f \in \mathbb{Z}$ и

$$f \leqslant x < f + 1.$$

Стоит упомянуть функцию **математического округления**, то есть округления к ближайшему целому. Она определяется как число r, такое, что

$$|x - r| = \min_{z \in \mathbb{Z}} |x - z|.$$

При этом, если этому соотношению отвечают два числа r_1 и r_2, возможны разные подходы. Один из них предполагает выбор большего из чисел r_1 и r_2, а другой — ближайшего чётного числа.

Однако функция математического округления будет рассматриваться лишь как конечная операция округления числа, точного до некоторого знака после запятой. На промежуточных этапах вычислений будет применяться округление к нулю (усечение), при котором лишние знаки отбрасываются. Все рассмотренные функции округления до целого очевидным образом распространяются на случаи округления до n-го знака.

Понимание и правильное применение методов округления позволяет минимизировать накопление ошибок и обеспечить высокую точность вычислений в алгоритмах с произвольной точностью.

1.1.4 Свойства операции округления, связанные с неравенствами

В дальнейшем нам потребуется исследовать округлённые значения некоторых величин a и b, для которых установлено отношение порядка. Без ограничения общности положим, что $a \leqslant b$. Тогда из определения функции округления вниз следует, что

$$\lfloor a \rfloor \leqslant a \leqslant b < \lfloor b \rfloor + 1, \tag{1.5}$$

откуда

$$\lfloor a \rfloor \leqslant \lfloor b \rfloor. \tag{1.6}$$

Действительно, предположим противное, то есть что

$$\lfloor a \rfloor > \lfloor b \rfloor.$$

Тогда $\lfloor a \rfloor \geqslant \lfloor b \rfloor + 1$, так как $\lfloor a \rfloor$ и $\lfloor b \rfloor$ — целые числа. Но полученный вывод противоречит условию (1.5). Значит, наше предположение неверно, и верно обратное неравенство (1.6).

Аналогично, для функции округления вверх имеем

$$\lceil a \rceil - 1 < a \leqslant b \leqslant \lceil b \rceil,$$

откуда

$$\lfloor a \rfloor \leqslant \lfloor b \rfloor, \tag{1.7}$$

поскольку $\lceil a \rceil$ и $\lceil b \rceil$ — целые числа. Неравенства (1.6) и (1.7) сохраняют свой вид и при более строгом условии $a < b$.

Рассмотрим сумму чисел a и b. Если сложить неравенства

$$\lfloor a \rfloor \leqslant a, \qquad \lfloor b \rfloor \leqslant b,$$

то получим

$$\lfloor a \rfloor + \lfloor b \rfloor \leqslant a + b.$$

По определению, $\lfloor a + b \rfloor$ — наибольшее целое число, не превосходящее сумму $a + b$, а значит,

$$\lfloor a \rfloor + \lfloor b \rfloor \leqslant \lfloor a + b \rfloor.$$

Аналогично для округления вверх имеем

$$a + b \leqslant \lceil a \rceil + \lceil b \rceil.$$

При этом $\lceil a+b \rceil$ — наименьшее целое число, не меньшее суммы $a+b$, а значит,

$$\lceil a + b \rceil \leqslant \lceil a \rceil + \lceil b \rceil.$$

Полученные свойства операции округления могут быть применены в различных вычислительных задачах.

1.1.5 Оценка порядка числа

В дальнейшем перед нами возникнет необходимость оценки модуля числа, что позволит сократить время и трудозатраты, отводимые на вычисления, не требующие большой точности. Этого можно добиться путём оценки порядков величин, входящих в то или иное выражение. **Порядок** величины — это число, которое указывает, сколько раз эту величину можно представить в виде степени числа 10. Экспонента числа, записанного в научной форме, есть ни что иное, как его порядок. Заметим, что число 0 не имеет порядка по определению, а порядки чисел a и $-a$ совпадают. В дальнейшем, говоря об оценке порядка числа, будем подразумевать оценку порядка его абсолютной величины.

При оценке порядка числа a удобно пользоваться его научной формой (1.1). Рассмотрим выражение:

$$\lg|a| = \lg\bigl(|\alpha|\cdot 10^n\bigr) = \lg|\alpha| + \lg 10^n = \lg|\alpha| + n.$$

Заметим, что

$$1 \leqslant |\alpha| < 10 \quad \Rightarrow \quad 0 \leqslant \lg|\alpha| < 1.$$

Таким образом, целая часть числа $\lg|a|$ равна n, а дробная — $\lg|\alpha|$. Этот логарифм удобно использовать для оценки порядка числа a.

Нижняя оценка необходима, когда требуется округлить модуль числа a в меньшую сторону. В таком случае примем её равной

$$\bigl\lfloor \lg|a| \bigr\rfloor = n.$$

Если же нас интересует обратная ситуация и требуется верхняя оценка порядка, разумно считать её равной

$$\bigl\lceil \lg|a| \bigr\rceil = \begin{cases} n, & \text{если } |\alpha| = 1, \\ n+1, & \text{если } 1 < |\alpha| < 10. \end{cases}$$

Итак, поскольку

$$\bigl\lfloor \lg|a| \bigr\rfloor \leqslant \lg|a| \leqslant \bigl\lceil \lg|a| \bigr\rceil,$$

то

$$10^{\lfloor \lg|a| \rfloor} \leqslant |a| \leqslant 10^{\lceil \lg|a| \rceil},$$

что соответствует интуитивным представлениям о нижней и верхней оценках величины числа. Для краткости дальнейших рассуждений введём следующие функции:

$$\Theta(x) = 10^{\lceil \lg |x| \rceil}, \tag{1.8}$$

$$\Omega(x) = 10^{\lfloor \lg |x| \rfloor}. \tag{1.9}$$

Будем говорить, что $\Theta(x)$ — верхняя оценка числа x по порядку, а $\Omega(x)$ — нижняя. Важно ещё раз подчеркнуть, что мы имеем в виду оценку модуля числа, а не его самого. Так, если

$$\Omega(x) \leqslant |x| \leqslant \Theta(x),$$

то для $x < 0$

$$x \leqslant \Omega(x) \leqslant \Theta(x),$$

что может привести к неверным толкованиям.

Возможны случаи, когда число a, порядок которого следует оценить, представляет собой сложное выражение. Однако вычислять его значение в точности нецелесообразно, поскольку, как было сказано ранее, оценка порядка призвана сократить количество вычислений. В таком случае достаточно узнать только первую значащую цифру числа. Это позволит определить её позицию в числе, а значит, найти порядок по формуле (1.3). При $\lambda = 1$ получаем $n = k$.

Таким образом, оценка порядка числа с использованием его логарифма является эффективным инструментом для оптимизации вычислений, особенно при работе с числами большой или малой величины.

1.1.6 Абсолютная погрешность и её влияние на точность вычислений

При вычислении приближённых значений величин нам придётся повсеместно прибегать к округлениям. Можно выделить два случая. В первом из них для округления нужно в точности определить цифру, следующую за последней интересующей нас, поскольку она может оказывать влияние на результат. Во втором случае нет необходимости в нахождении большего количества цифр, чем требуется. В качестве примера можно привести округление к ближайшему целому для первого случая и округление путём отбрасывания ненужных цифр

— для второго.

Точность вычислений удобно измерять абсолютной систематической погрешностью величины. Так, если мы хотим получить значение величины a с точностью до k-го разряда, это означает, что абсолютная погрешность Δa этой величины должна удовлетворять неравенству

$$\Delta a < \frac{1}{2} \cdot 10^k,$$

если округление зависит от следующей цифры, и

$$\Delta a < 10^k,$$

если не зависит. Итоговое значение будет иметь вид $a \pm \Delta a$.

На практике необходимо обеспечить выполнение следующего условия: все вычисленные цифры приближения числа a должны совпадать с соответствующими цифрами истинного значения этой величины, а последняя — быть увеличенной на 1, если того требуют правила округления. Например, если истинное значение величины имеет вид $a = 0{,}12345\ldots$, то эта же величина, округлённая до тысячных, должна иметь вид $a \approx 0{,}123$, а округлённая до десятитысячных — $a \approx 0{,}1235$, если речь идёт об округлении до ближайшего числа.

Как видно, для достижения заданной точности погрешность не должна превышать $5 \cdot 10^{k-1}$ при округлении к ближайшему числу. Для удобства будем считать, что во всех случаях

$$\Delta a = 10^{k-1}.$$

В дальнейшем будем называть величину $\kappa = k - 1$ истинной точностью, а k — требуемой.

Важно обратить внимание на то, как ведут себя погрешности чисел при выполнении над ними арифметических и других операций. Пусть имеется набор величин x_1, …, x_n, обладающих собственными систематическими погрешностями, и функция $f = f(x_1, \ldots, x_n)$. Тогда абсолютная систематическая погрешность функции f может быть найдена как

$$\Delta f = \sum_{i=1}^{n} \left| \frac{\partial f}{\partial x_i} \right| \cdot \Delta x_i.$$

Таким образом, при суммировании и вычитании погрешности результата операции выглядят следующим образом:

$$\Delta(a+b) = \Delta(a-b) = \Delta a + \Delta b.$$

Эта формула распространяется на любое число слагаемых:

$$\Delta\left(\sum_{i=1}^{N} \pm a_i\right) = \sum_{i=1}^{N} \Delta a_i. \tag{1.10}$$

В случае умножения и деления будем иметь

$$\Delta(a \cdot b) = |a| \cdot \Delta b + |b| \cdot \Delta a, \tag{1.11}$$

и

$$\Delta\left(\frac{a}{b}\right) = \frac{\Delta a}{|b|} + \frac{|a| \cdot \Delta b}{b^2}, \tag{1.12}$$

соответственно. Для возведения числа a в степень n, если показатель степени не обладает собственной погрешностью, получим

$$\Delta(a^n) = \left|n \cdot a^{n-1}\right| \cdot \Delta a, \tag{1.13}$$

а если имеется погрешность Δn, то

$$\Delta(a^n) = \left|n \cdot a^{n-1}\right| \cdot \Delta a + \left|a^n \cdot \ln a\right| \cdot \Delta n.$$

Понимание и правильная оценка погрешностей являются критически важными для разработки алгоритмов, обеспечивающих высокую точность вычислений.

1.2 Оптимизация вычислений и оценка точности при использовании ряда Тейлора

1.2.1 Ряд Тейлора

Для численного приближения значения элементарной функции $f(x)$ в точке x удобно использовать **формулу Тейлора**:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (x-a)^n, \tag{1.14}$$

где $f^{(n)}(a)$ — n-я производная функции $f(x)$ в точке a. Член ряда Тейлора под номером n будем обозначать через R_n:

$$R_n = \frac{f^{(n)}(a)}{n!} (x-a)^n.$$

Хорошим упрощением расчётов станет такой выбор числа a, чтобы значение производной $f^{(n)}(a)$ не зависело от n. Тогда мы сможем рассматривать эту величину как константу:

$$f(x) = f^{(i)}(a) \cdot \sum_{n=0}^{\infty} \frac{(x-a)^n}{n!},$$

где i — любой номер из набора индексов членов ряда. Однако это не всегда возможно и не обязательно.

Если $|x| < 1$, разумно принять $a = 0$ в тех случаях, когда f дифференцируема в этой точке. Тогда формула Тейлора перейдёт в формулу Маклорена:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{n!} x^n.$$

1.2.2 Сходимость ряда и радиус сходимости

Важным аспектом работы со степенными рядами является необходимость обеспечить сходимость ряда, иначе получить определённое значение не удаст-

ся. Необходимым условием сходимости ряда является следующее:

$$\lim_{n\to\infty} R_n = 0. \tag{1.15}$$

Однако на практике помимо самой сходимости важна её скорость: если ряд сходится медленно, вычисление значения частичной суммы ряда займёт недопустимо большое время. Для нас будет важно, чтобы последовательность $\{|R_n|\}$ монотонно убывала, то есть

$$\forall n\big(|R_{n+1}| \leqslant |R_n|\big) \tag{1.16}$$

(подразумевается, что $n \in \mathbb{Z}_+$). Если это условие выполняется для ряда с неотрицательными членами ($R_n \geqslant 0$), то по признаку Д'Аламбера он сходится:

$$\lim_{n\to\infty} \frac{R_{n+1}}{R_n} < 1.$$

В этом случае проверка условия (1.15) не требуется. Для знакочередующихся рядов, то есть рядов с членами вида $R_n = (-1)^n \cdot r_n$, по признаку Лейбница требуется выполнение обоих условий (1.15) и (1.16). Иными словами, последовательность $\{|R_n|\}$ должна монотонно стремиться к нулю.

Если члены ряда убывают достаточно быстро, то количество членов, необходимое для расчёта частичной суммы с требуемой точностью, будет невелико. Это может также пригодиться, если возникнет необходимость оценить величину суммы ряда до его непосредственного вычисления. Однако в некоторых случаях придётся прибегать к более жёстким условиям для обеспечения высокой скорости сходимости ряда.

Наконец, важно рассмотреть понятие радиуса сходимости степенного ряда:

$$f(x) = \sum_{n=0}^{\infty} c_n \cdot z^n,$$

где $z = x - a$. **Радиусом сходимости** называют такую величину R, что рассматриваемый ряд сходится на интервале

$$K = \{z \mid |z| < R\}.$$

Если существует конечный или бесконечный предел $\lim\limits_{n\to\infty}\left|\dfrac{c_n}{c_{n+1}}\right|$, то

$$R = \lim_{n\to\infty}\left|\frac{c_n}{c_{n+1}}\right|.$$

В случае, если этот предел не существует, можно воспользоваться формулой Коши-Адамара:

$$\frac{1}{R} = \overline{\lim_{n\to\infty}}\sqrt[n]{|c_n|}.$$

В дальнейшем мы будем учитывать радиус сходимости при выборе методов разложения функций в ряды для обеспечения сходимости и требуемой точности вычислений.

1.2.3 Оптимизация вычислений с использованием ряда Тейлора

Суммирование членов ряда должно осуществляться конечное число раз. Обозначим это число через N , а индекс, начиная с которого производится суммирование, — через $n_{\min}$. Тогда последний член ряда, который войдёт в необходимую нам частичную сумму, будет иметь номер

$$m = n_{\min} + N - 1.$$

При вычислении каждого слагаемого необходимо учитывать истинную точность κ. Если погрешность вычисления функции составляет

$$\Delta f(x) = 10^{\kappa},$$

то погрешность каждого члена ряда удобно принять равной

$$\Delta R \equiv \Delta R_n = \frac{10^{\kappa}}{N}. \tag{1.17}$$

Такой подход называют **принципом равных влияний**. В этом случае суммарная погрешность совпадёт с необходимой:

$$\Delta f(x) = \sum_{n=n_{\min}}^{m} \Delta R_n = \frac{10^{\kappa}}{N}\cdot N = 10^{\kappa}.$$

Следовательно, прежде чем начать вычисление отдельных слагаемых ряда Тейлора, необходимо знать их точное количество.

Остановить вычисление новых членов следует тогда, когда следующие слагаемые перестанут вносить значимый вклад в сумму с заданной точностью κ. Первым таким слагаемым станет R_M:

$$|R_M| = \left| \frac{f^{(M)}(a)}{M!} (x-a)^M \right| < \Delta R,$$

где $M = n_{\min} + N$. Когда такое слагаемое будет найдено, можно определить число N:

$$N = M - n_{\min}.$$

Чтобы не выполнять вычисления дважды (сначала для определения числа N, а затем для получения значений слагаемых с необходимой точностью), можно ограничиться грубыми оценками величин, входящих в формулы. Выведем методику оценки порядка в общем виде. Пусть

$$|R_n| = \frac{\prod_i |\nu_i(n)|}{\prod_j |\delta_j(n)|}$$

Тогда, чтобы оценить эту величину сверху, округлим числитель в большую сторону, а знаменатель — в меньшую:

$$|R_n| \leqslant \frac{\prod_i \Theta\big(|\nu_i(n)|\big)}{\prod_j \Omega\big(|\delta_j(n)|\big)} = \frac{10^{\sum_i \lceil \lg |\nu_i(n)| \rceil}}{10^{\sum_j \lfloor \lg |\delta_j(n)| \rfloor}} = 10^{\xi},$$

где

$$\xi = \sum_i \big\lceil \lg |\nu_i(n)| \big\rceil - \sum_j \big\lfloor \lg |\delta_j(n)| \big\rfloor.$$

Величину ξ можно рассматривать как верхнюю оценку порядка члена R_n, то есть номера разряда его первой значащей цифры.

Аналогично можно поступить и с оценкой снизу для ΔR:

$$\Delta R \geqslant \frac{10^{\kappa}}{\Theta(N)} = 10^{\chi},$$

где

$$\chi = \kappa - \lceil \lg N \rceil.$$

По своему смыслу, χ — это нижняя оценка номера разряда последней значащей цифры, которую необходимо вычислить для R_n.

Теперь, если $10^{\xi} < 10^{\chi}$, то $|R_M| < \Delta R$ гарантировано. Таким образом, необходимо, чтобы выполнялось неравенство

$$\xi < \chi. \tag{1.18}$$

Далее условимся считать, что

$$\Delta R = 10^{\chi}.$$

Более строго эта величина называется границей абсолютной погрешности.

Таким образом, разработанная методика позволяет эффективно оценивать необходимое число членов ряда Тейлора для достижения заданной точности, что существенно оптимизирует вычисления и повышает их эффективность.

1.2.4 Оптимизация сходимости ряда через нормализацию аргумента

Если член ряда имеет вид

$$(-1)^n \cdot \frac{x^n}{n!}$$

и при этом $|x| \leqslant 1$, то переживать о скорости его сходимости не придётся: знаменатель $n!$ возрастает достаточно быстро, и с его ростом значения членов убывают. Однако члены некоторых рядов убывают гораздо медленнее. Например, если

$$|R_n| = \frac{|x|^{2n+1}}{2n+1}. \tag{1.19}$$

В таком случае члены ряда практически не убывают при значениях $|x|$, близких к единице. Выходом из положения может служить нахождение множества значений x, при которых скорость сходимости будет оптимальна.

Рассмотрим поставленную задачу на примере ряда с членами (1.19). Достаточно быстрая сходимость такого ряда будет обеспечена, если каждый его член по модулю будет меньше предыдущего хотя бы на один порядок. Однако отслеживать выполнение такого условия во многих случаях трудоёмкая задача.

Поэтому примем более строгое, но более ясное правило:

$$|R_{n+1}| \leqslant \frac{1}{10} \cdot |R_n|. \tag{1.20}$$

Для упрощения оценок можно пренебречь изменением знаменателей при больших n, так как $2(n+1)+1 \approx 2n+1$. Тогда неравенство упростится до:

$$|x|^{2(n+1)+1} \leqslant \frac{1}{10} \cdot |x|^{2n+1},$$

что эквивалентно

$$|x|^2 \leqslant \frac{1}{10}.$$

Отсюда следует

$$|x| \leqslant \frac{\sqrt{10}}{10}.$$

Это неравенство задаёт интервал значений величины x, допустимых для оптимальной скорости сходимости ряда. Если x лежит вне этого диапазона, то его следует нормализовать, то есть представить в таком виде, чтобы в конечном счёте в качестве аргумента в разложении функции фигурировала величина, принадлежащая множеству

$$\left[-\frac{\sqrt{10}}{10}; \frac{\sqrt{10}}{10}\right].$$

Более строго **нормализацией аргумента** называется преобразование исходной функции так, чтобы её аргумент находился в требуемом диапазоне.

Таким образом, нормализация позволяет сократить количество необходимых членов ряда для достижения заданной точности, что повышает эффективность вычислений.

1.2.5 Сокращение вычислительных затрат при нормализации аргумента ряда

Оценка количества членов ряда, необходимых для достижения требуемой точности, если они удовлетворяют условию (1.20), может быть произведена

особым образом. Рассмотрим ряд

$$f(x) = \sum_{n=n_{\min}}^{\infty} R_n.$$

Частичную сумму из N членов этого ряда будем обозначать через

$$f_m(x) = \sum_{n=n_{\min}}^{m} R_n,$$

где $m = n_{\min} + N - 1$. **Остатком ряда** ρ_m называется величина

$$\rho_m = f(x) - f_m(x) = \sum_{n=m+1}^{\infty} R_n.$$

Таким образом, необходимо остановить вычисления тогда, когда абсолютная величина остатка ряда станет меньше заданной погрешности:

$$|\rho_m| < \Delta f(x). \tag{1.21}$$

При исследовании модуля остатка ряда удобно применить неравенство треугольника:

$$\forall a \forall b \big(|a + b| \leqslant |a| + |b| \big)$$

(подразумевается, что $a, b \in \mathbb{R}$). Тогда

$$|\rho_m| = \left| \sum_{n=m+1}^{\infty} R_n \right| \leqslant \sum_{n=m+1}^{\infty} |R_n|.$$

Выразим $|R_n|$ через $|R_{m+1}|$ с использованием условия (1.20):

$$|R_n| \leqslant \left(\frac{1}{10} \right)^{n-(m+1)} |R_{m+1}|,$$

где $n \geqslant m + 1$. Тогда для остатка ряда имеем

$$|\rho_m| \leqslant |R_{m+1}| \cdot \sum_{n=m+1}^{\infty} \left(\frac{1}{10} \right)^{n-(m+1)}. \tag{1.22}$$

Теперь сумму в правой части неравенства можно упростить следующим образом:

$$\sum_{n=m+1}^{\infty} \left(\frac{1}{10}\right)^{n-m-1} = \sum_{n=0}^{\infty} \left(\frac{1}{10}\right)^{n}.$$

Это есть ни что иное, как сумма бесконечной геометрической прогрессии с первым членом $b = 1$ и знаменателем $q = \frac{1}{10}$. Её значение можно вычислить по известной формуле:

$$\sum_{n=0}^{\infty} \left(\frac{1}{10}\right)^{n} = \frac{b}{1-q} = \frac{10}{9}.$$

Если подставить это значение в неравенство (1.22), получим

$$|\rho_m| \leqslant |R_{m+1}| \cdot \frac{10}{9}.$$

Если необходимо вычислить значение $f(x)$ с истинной точностью κ, то условие (1.21) можно заменить более жёстким:

$$|R_{m+1}| \cdot \frac{10}{9} < 10^{\kappa} \quad \Leftrightarrow \quad |R_{m+1}| < \frac{9}{10} \cdot 10^{\kappa}.$$

Для удобства можно принять

$$|R_{m+1}| < 10^{\kappa-1} \quad \Leftrightarrow \quad |R_{m+1}| < \Delta f(x).$$

Будем пользоваться этим неравенством для поиска числа m, а затем определим необходимое количество членов ряда N:

$$N = m - n_{\min} + 1.$$

Оценку члена R_{m+1} можно выполнить предложенным ранее способом.

Полученный метод позволяет эффективно определять необходимое число членов ряда для достижения заданной точности, что повышает эффективность вычислительных алгоритмов.

1.2.6 Оптимальный поиск количества подлежащих вычислению членов ряда

Простейший способ найти N, то есть количество подлежащих вычислению членов ряда $f(x)$, — последовательно оценивать порядок членов ряда, начиная с наименьшего по индексу, пока не встретится член с индексом M, который удовлетворит условию

$$|R_M| < \Delta R \tag{1.23}$$

или

$$|R_M| < \Delta f(x) \tag{1.24}$$

в случае нормализованного аргумента. Однако в вычислениях, требующих большой точности, число M может быть достаточно велико, что сильно замедлит расчёты. На помощь может прийти метод бисекции. Но заметим, что для индекса n любого члена ряда верно, что $n \in [n_{\min}, +\infty)$, то есть множество индексов неограничено сверху. В таком случае для эффективного поиска M нужно ограничить диапазон возможных значений.

Прежде всего следует проверить, что член под номером $n_{\min}$ по модулю не меньше погрешности ΔR или $\Delta f(x)$. В противном случае величина k — номер младшего разряда, подлежащего вычислению, — задана некорректно, и вычисления не могут быть произведены. Если же такой ошибки не произойдёт, следует перейти к следующим действиям. Для этого разделим множество значений индексов на подмножества, в которых будем осуществлять поиск. Первым из них станет $[n_{\min}, 2 \cdot n_{\min}]$, если $n_{\min} > 0$, и $[0, 1]$, если $n_{\min} = 0$. Проверим правую границу подмножества — $2 \cdot n_{\min}$. Если член с этим индексом не удовлетворяет условию (1.23) или (1.24) в соответствующем случае, перейдём к следующему подмножеству: $[2 \cdot n_{\min}, 4 \cdot n_{\min}]$, то есть бывшая правая граница станет левой, а новая правая снова будет вдвое больше новой левой. Будем продолжать это до тех пор, пока не будет найдено подмножество, правая граница которого соответствует члену, меньшему погрешности.

Когда необходимое подмножество $[l, u]$ определено, можно перейти к бисекции. Выберем средний элемент:

$$M = \left\lfloor \frac{l + u}{2} \right\rfloor.$$

Если член с индексом M по модулю меньше погрешности, то перейдём к множеству $[l, M-1]$, иначе — к множеству $[M+1, u]$. Повторять эти действия будем до тех пор, пока левая граница не превысит правую. Это будет значить, что искомый минимальный индекс M, при котором $|R_M| < \Delta R$, найден и равен текущему значению левой границы l.

Данный алгоритм позволяет эффективно определить минимальное количество членов ряда, необходимых для достижения заданной точности, что существенно оптимизирует вычисления и снижает время выполнения алгоритмов с использованием степенных рядов.

1.2.7 Вычисление порядка бесконечной суммы ряда

Задача вычисления порядка может быть сведена к вычислению исследуемой величины $f(x)$ до первой значащей цифры ($\lambda = 1$). Однако при рассмотрении рядов мы опирались на точность, выраженную через номер младшего разряда k, значение которого необходимо вычислить. При этом, чтобы такую точность выразить через количество значащих цифр, необходимо заранее знать порядок n величины $f(x)$: $k = n$.

Чтобы решить поставленную задачу, вместо порядка величины подставим в формулу (**??**) при $\lambda = 1$ нижнюю оценку порядка n_f. Так мы получим заниженный номер младшего разряда $k_f = n_f$, а значит, вычислим больше значащих цифр.

Рассмотрим ряды, для членов которых выполняется условие (1.20). Выразим n-й член через член с индексом $n_{\min}$:

$$|R_{\text{n}}| \leqslant |R_{n_{\min}}| \cdot \left(\frac{1}{10}\right)^{n-n_{\min}}.$$

Используя известные преобразования, получим для остатка ρ_m соотношение:

$$|\rho_m| \leqslant |R_{n_{\min}}| \cdot \sum_{n=m+1}^{\infty} \left(\frac{1}{10}\right)^{n-n_{\min}} = \frac{|R_{n_{\min}}|}{9} \cdot \left(\frac{1}{10}\right)^{m+1-n_{\min}}.$$

Положим $m = n_{\min}$. Тогда

$$|\rho_{n_{\min}}| \leqslant \frac{|R_{n_{\min}}|}{90}. \tag{1.25}$$

Далее заметим, что

$$f(x) = R_{n_{\min}} + \rho_{n_{\min}} = R_{n_{\min}} - (-\rho_{n_{\min}}).$$

Воспользуемся леммой $|a-b| \geqslant \big||a|-|b|\big|$ (она является следствием неравенства треугольника):

$$|f(x)| = |R_{n_{\min}} - (-\rho_{n_{\min}})| \geqslant \big||R_{n_{\min}}| - |\rho_{n_{\min}}|\big|. \tag{1.26}$$

При этом из (1.25) следует, что $|\rho_{n_{\min}}| \leqslant |R_{n_{\min}}|$, то есть $|R_{n_{\min}}| - |\rho_{n_{\min}}| \geqslant 0$. Поэтому неравенство можно записать в более удобном виде:

$$|f(x)| \geqslant |R_{n_{\min}}| - |\rho_{n_{\min}}|.$$

Вновь воспользуемся неравенством (1.25), чтобы избавиться от величины $|\rho_{n_{\min}}|$:

$$|f(x)| \geqslant |R_{n_{\min}}| - \frac{|R_{n_{\min}}|}{90} = \frac{89}{90} \cdot |R_{n_{\min}}|.$$

Как было отмечено ранее, вычисление порядка легко представить через логарифмирование:

$$\lg|f(x)| \geqslant \lg\frac{89}{90} + \lg|R_{n_{\min}}|,$$

откуда

$$\lfloor \lg|f(x)| \rfloor \geqslant \left\lfloor \lg\frac{89}{90} + \lg|R_{n_{\min}}| \right\rfloor \geqslant \left\lfloor \lg\frac{89}{90} \right\rfloor + \lfloor \lg|R_{n_{\min}}| \rfloor = \lfloor \lg|R_{n_{\min}}| \rfloor - 1.$$

Величина $n_f = \lfloor \lg|R_{n_{\min}}| \rfloor - 1$ является заниженной оценкой порядка $f(x)$.

Таким образом, для вычисления порядка бесконечной суммы ряда рассмотренного типа можно ограничиться вычислением первого его члена до разряда с индексом n_f.

1.3 Численное приближение иррациональных констант

1.3.1 Приближение числа e

Разложение числа Эйлера e в ряд имеет вид

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}.$$

Члены этого ряда удовлетворяют условию (1.15), а значит, ряд сходится и притом достаточно быстро. Разряд первой значащей цифры члена R_n можно оценить сверху как

$$\xi = -\lfloor \lg n! \rfloor.$$

Чтобы оценить количество необходимых членов, будем пользоваться неравенством (1.18).

1.4 Численное приближение логарифмических функций

1.4.1 Приближение натурального логарифма нормализованного аргумента

Ряд Тейлора для натурального логарифма в общем случае обладает очень медленной сходимостью и поэтому не применим для расчётов на практике. Рассмотрим два ряда Маклорена:

$$\ln(1+x) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{n} \cdot x^n,$$

$$\ln(1-x) = -\sum_{n=1}^{\infty} \frac{1}{n} \cdot x^n.$$

Эти ряды сходятся при $|x| < 1$. Обратим внимание на свойство разности логарифмов:

$$\ln(1+x) - \ln(1-x) = \ln\left(\frac{1+x}{1-x}\right).$$

Итак,

$$\ln\left(\frac{1+x}{1-x}\right) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{n} \cdot x^n + \sum_{n=1}^{\infty} \frac{1}{n} \cdot x^n = \\ = \sum_{n=1}^{\infty} \left(\frac{(-1)^{n-1}}{n} \cdot x^n + \frac{1}{n} \cdot x^n \right) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1}+1}{n} \cdot x^n. \quad (1.27)$$

Заметим, что при чётных значениях n члены ряда обращаются в ноль:

$$(-1)^{n-1} + 1 = -1 + 1 = 0.$$

Если же n нечётно, то

$$(-1)^{n-1} + 1 = 1 + 1 = 2.$$

Таким образом, выражение (1.27) можно переписать в виде

$$\ln\left(\frac{1+x}{1-x}\right) = 2 \cdot \sum_{n=0}^{\infty} \frac{x^{2n+1}}{2n+1}.$$

Это разложение называется рядом Меркатора, усовершенствованным Джеймсом Грегори. Благодаря знаменателю $2n+1$ его члены убывают несколько быстрее, чем члены ряда Маклорена. Ряд Меркатора также сходится для значений $|x| < 1$.

Допустим, нам необходимо найти логарифм произвольного числа z. Тогда

$$z = \frac{1+x}{1-x} \quad \Rightarrow \quad x = \frac{z-1}{z+1}.$$

Отсюда видно, что $|x| < 1$ при любом положительном z, однако при больших значениях x (то есть значениях близких к единице) ряд сходится крайне медленно. Это происходит потому, что знаменатель $2n+1$ всё ещё возрастает медленно (гораздо медленнее, чем $n!$) и значительное убывание членов может обеспечить только числитель x^{2n+1}. Разумным будет исходить из принципа уменьшения порядка числителя как минимум на единицу при переходе от n-го члена к $(n+1)$-му, как было предложено ранее.

Итак, ранее мы пришли к выводу, что оптимальная скорость сходимости

подобных рядов обеспечивается при

$$|x| \leqslant \frac{\sqrt{10}}{10},$$

откуда для z имеем

$$\frac{11 - 2\sqrt{10}}{9} \leqslant z \leqslant \frac{11 + 2\sqrt{10}}{9}.$$

Для удобства перейдём к более приближённым неравенствам:

$$0{,}52 \leqslant z \leqslant 1{,}92. \tag{1.28}$$

Если число z удовлетворяет условию (1.28), можно перейти к оценке количества членов ряда, которые необходимо учесть для корректного приближения логарифма. Для этого порядок члена R_n оценивается, как

$$\xi = \lceil \lg |x|^{2n+1} \rceil - \lfloor \lg(2n+1) \rfloor.$$

Поскольку мы пока не знаем, с какой точностью должна быть вычислена величина x, достаточно узнать лишь её первую значащую цифру. При этом результат должен быть округлён вверх по модулю.

Когда будет определено количество членов, а значит, будет известна погрешность члена 10^{χ}, можно определить погрешность x:

$$\Delta R_n = \frac{\Delta x^{2n+1}}{2n+1} \quad \Leftrightarrow \quad \Delta x^{2n+1} = (2n+1) \cdot 10^{\chi} \quad \Leftrightarrow$$
$$\Leftrightarrow \quad (2n+1) \cdot |x|^{2n} \cdot \Delta x = (2n+1) \cdot 10^{\chi} \quad \Leftrightarrow \quad \Delta x = \frac{10^{\chi}}{|x|^{2n}}.$$

При этом $\Delta x > 10^{\chi}$ и точность, с которой должен быть вычислен x, можно принять равной χ.

1.4.2 Нормализация аргумента натурального логарифма

Рассмотрим случай, когда z лежит вне отрезка $[0{,}52;\, 1{,}92]$. Наиболее очевидным решением этой проблемы является поиск такого числа a, что

$$\zeta = z \cdot a$$

и ζ удовлетворяет условию (1.28). Тогда, ввиду свойства логарифма частного,

$$\ln z = \ln\left(\frac{\zeta}{a}\right) = \ln\zeta - \ln a.$$

Отсюда видно, что наиболее удобным значением числа a будет некоторая степень числа Эйлера: $a = e^n$. В этом случае

$$\ln z = \ln\zeta - n. \tag{1.29}$$

То есть задача нормализации аргумента z сводится к нахождению такого числа n, при котором число

$$\zeta = z \cdot e^n$$

лежит на отрезке $[0{,}52; 1{,}92]$.

Для любого положительного числа z существует вещественное число m, такое, что $z = e^m$. При этом, по свойству округления вниз,

$$\lfloor m \rfloor \leqslant m < \lfloor m \rfloor + 1 \quad \Rightarrow \quad e^{\lfloor m \rfloor} \leqslant e^m < e^{\lfloor m \rfloor + 1},$$

а значит, найдутся два целых числа l и $l+1$ таких, что

$$e^l \leqslant z < e^{l+1}. \tag{1.30}$$

Поделив это неравенство на e^l, получим

$$1 \leqslant z \cdot e^{-l} < e.$$

Выходит, что $z \cdot e^{-l}$ гарантированно лежит в интервале $[1; e)$, но может выйти за правый предел $1{,}92$ интересующего нас отрезка. Если $z \cdot e^{-l} \leqslant 1{,}92$, то положим $n = -l$. В противном случае, то есть при

$$1{,}92 < z \cdot e^{-l} < e,$$

разделим это неравенство на e:

$$\frac{1{,}92}{e} < z \cdot e^{-l-1} < 1.$$

Поскольку величина $\frac{1{,}92}{e}$ превосходит число $0{,}52$, то $z \cdot e^{-l-1}$ гарантированно лежит в заданном диапазоне $[0{,}52;\, 1{,}92]$. Итак, в этом случае $n = -l - 1$.

Если z меньше числа 0,52, то величина l отрицательна и является наибольшим из целых чисел, для которых верно

$$e^l \leqslant z.$$

Если же z больше числа 1,92, то l положительно и представляет собой наименьшее из целых чисел, для которых верно

$$z < e^{l+1}.$$

Будем пользоваться этим свойством для нахождения числа l по методике, аналогичной способу поиска количества членов ряда, необходимых для приближения. Для корректного сравнения с величиной z необходимо, чтобы числа e^{l+1} и e^l содержали столько же цифр после запятой, сколько их содержит z. При приближении числа e^l должно применяться округление вверх, а при приближении числа e^{l+1} — округление вниз.

Когда число l будет найдено, проверим условие $z \cdot e^{-l} < 1{,}92$. Для произведения $z \cdot e^{-l}$ необходимо вычислить два знака после запятой, а значит,

$$z \cdot \Delta e^{-l} = 10^{-2} \quad \Leftrightarrow \quad \Delta e^{-l} = \frac{10^{-2}}{z}.$$

В таком случае точность вычисления e^{-l} составляет $-2 - \lceil \lg z \rceil$.

После того, как n определено, необходимо узнать, с какой точностью должен быть вычислен аргумент ζ. Если истинная точность вычисления $\ln z$ равна κ, то, по формуле (1.29), имеем

$$\Delta \ln \zeta = \Delta \ln z = 10^{\kappa}.$$

Тогда

$$\Delta \ln \zeta = \frac{\Delta\zeta}{\zeta} \quad \Leftrightarrow \quad \Delta\zeta = \zeta \cdot 10^{\kappa} > \frac{10^{\kappa}}{2}.$$

Кроме того,

$$\Delta\zeta = z \cdot \Delta e^n \quad \Leftrightarrow \quad \Delta e^n = \frac{\zeta}{z} \cdot 10^{\kappa} > \frac{1}{2z} \cdot 10^{\kappa}.$$

Наконец, точность вычисления e^n примем равной

$$\kappa - \lceil \lg 2z \rceil,$$

а точность вычисления ζ — равной $\kappa - 1$.

Теперь нормализованный аргумент ζ может быть использован для вычисления натурального логарифма числа z по формуле (1.29).

1.4.3 Нижняя оценка порядка натурального логарифма

Обратим внимание на формулу (1.29). При $n = 0$, то есть если аргумент z изначально не требует нормализации, нижняя оценка порядка натурального логарифма равна порядку величины $2R_0$, где $R_0 = x$. То есть

$$n_f = \lfloor \lg 2|x| \rfloor - 1.$$

При этом величина x может быть вычислена до первой значащей цифры с отбрасыванием лишних знаков.

В том случае, если $n \neq 0$, то именно эта величина n определяет порядок логарифма $\ln z$. При этом логарифм числа ζ не превышает единицы по модулю, а значит, не может изменить порядок $\ln z$ более чем на единицу. Тогда можно заключить, что

$$n_f = \lfloor \lg n \rfloor - 1.$$

1.4.4 Приближение логарифма с произвольным основанием

Если необходимо вычислить значение логарифма числа z по основанию $a \neq e$, то проще всего воспользоваться уже реализованным методом вычисления натурального логарифма. Обратимся к формуле перехода к новому основанию:

$$\log_a z = \frac{\log_b z}{\log_b a}.$$

Если взять $b = e$, то получим

$$\log_a z = \frac{\ln z}{\ln a}. \tag{1.31}$$

По формуле для абсолютной погрешности частного имеем:

$$\Delta \log_a z = \left|\frac{1}{\ln a}\right| \Delta \ln z + \left|\frac{\ln z}{(\ln a)^2}\right| \Delta \ln a.$$

Руководствуясь принципом равных влияний, приравняем вклады погрешностей:

$$\left|\frac{1}{\ln a}\right| \Delta \ln z = \left|\frac{\ln z}{(\ln a)^2}\right| \Delta \ln a = \frac{\Delta \log_a z}{2}.$$

Отсюда получаем:

$$\Delta \ln z = \frac{|\ln a|}{2} \Delta \log_a z, \qquad \Delta \ln a = \frac{|\ln a|^2}{2|\ln z|} \Delta \log_a z.$$

Пусть истинная точность, с которой должен быть вычислен логарифм $\log_a z$, равна κ. Тогда точность вычисления $\ln z$ оценивается как:

$$\kappa + \lfloor \lg \ln a \rfloor - 1,$$

а точность вычисления $\ln a$ как:

$$\kappa + \lfloor \lg(\ln^2 a) \rfloor - \lceil \lg \ln z \rceil - 1.$$

Особое внимание следует обратить на случай вычисления десятичного логарифма $\lg z$, то есть когда $a = 10$. В этом случае для $\ln z$ имеем точность

$$\kappa - 1,$$

а для $\ln 10$ — точность

$$\kappa - \lceil \lg \ln z \rceil - 1.$$

1.4.5 Нижняя оценка порядка логарифма с произвольным основанием

Оценим порядок n величины $\log_a z$. Исходя из формулы (1.31), имеем:

$$n = \lfloor \lg \ln z - \lg \ln a \rfloor \geqslant \lfloor \lg \ln z \rfloor + \lfloor -\lg \ln a \rfloor.$$

Заметим, что

$$\lfloor -\lg\ln a\rfloor \leqslant -\lg\ln a < \lfloor -\lg\ln a\rfloor + 1 \quad \Leftrightarrow$$
$$\Leftrightarrow \quad -\lfloor -\lg\ln a\rfloor - 1 < \lg\ln a \leqslant -\lfloor -\lg\ln a\rfloor.$$

Отсюда следует, что $\lfloor -\lg\ln a\rfloor = -\lceil\lg\ln a\rceil$. Таким образом, можем записать:

$$n \geqslant \lfloor\lg\ln z\rfloor - \lceil\lg\ln a\rceil.$$

Учитывая, что $\lceil\lg\ln a\rceil \leqslant \lfloor\lg\ln a\rfloor + 1$, получаем:

$$n \geqslant \lfloor\lg\ln z\rfloor - \lfloor\lg\ln a\rfloor - 1.$$

Итак, нижняя оценка порядка логарифма с произвольным основанием равна

$$n_f = \lfloor\lg\ln z\rfloor - \lfloor\lg\ln a\rfloor - 1.$$

При этом в случае десятичного логарифма

$$n_f = \lfloor\lg\ln z\rfloor - 1.$$

1.5 Численное приближение функций, связанных с возведением в степень

1.5.1 Приближение экспоненты

Экспонентой числа x называют функцию e^x. Разложим её в ряд Маклорена:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

Оценить порядок члена этого ряда можно, как

$$\lceil\lg x^n\rceil - \lfloor\lg n!\rfloor.$$

Несмотря на то, что ряд Маклорена для экспоненты сходится при любом x, на практике удобно использовать его только для значений $|x| < 1$, поскольку

для них ряд сходится наиболее быстро. В таком случае заметим, что

$$0 \leqslant x - \lfloor x \rfloor < 1.$$

Тогда, положив

$$x = \lfloor x \rfloor + \big(x - \lfloor x \rfloor\big),$$

получим

$$e^x = e^{\lfloor x \rfloor} \cdot e^{x - \lfloor x \rfloor}.$$

При этом множитель $e^{\lfloor x \rfloor}$ будем вычислять методом бинарного возведения в степень, а $e^{x-\lfloor x \rfloor}$ — с помощью ряда Маклорена.

Рассмотрим случай $x > 1$. Пусть необходимо вычислить e^x до разряда с номером κ. Тогда

$$\Delta e^x = 10^\kappa \quad \Rightarrow \quad e^{\lfloor x \rfloor} \cdot \Delta e^{x - \lfloor x \rfloor} + e^{x - \lfloor x \rfloor} \cdot \Delta e^{\lfloor x \rfloor} = 10^\kappa,$$

откуда, по принципу равных влияний, получаем

$$\Delta e^{x - \lfloor x \rfloor} = \frac{10^\kappa}{2e^{\lfloor x \rfloor}}, \qquad \Delta e^{\lfloor x \rfloor} = \frac{10^\kappa}{2e^{x - \lfloor x \rfloor}}.$$

Заметим, что

$$1 \leqslant e^{x - \lfloor x \rfloor} < e.$$

Тогда можно упростить выражение для $\Delta e^{\lfloor x \rfloor}$ следующим образом:

$$\Delta e^{\lfloor x \rfloor} > \frac{10^\kappa}{2e}.$$

В таком случае число e для бинарного возведения в степень с целым показателем $\lfloor x \rfloor$ должно быть вычислено с погрешностью

$$\Delta e = \frac{10^\kappa}{2e \cdot \lfloor x \rfloor \cdot e^{\lfloor x \rfloor - 1}}.$$

Учитывая, что $e < 3$, получаем

$$e^{\lfloor x \rfloor - 1} \leqslant 3^{\lfloor x \rfloor - 1}.$$

Наконец,

$$\Delta e > \frac{10^\kappa}{2e \cdot \lfloor x \rfloor \cdot 3^{\lfloor x \rfloor - 1}}$$

и точность числа e должна составлять

$$\kappa - 1 - \lceil \lg \lfloor x \rfloor \rceil - \lceil \lg 3^{\lfloor x \rfloor - 1} \rceil.$$

Для величины $\Delta e^{x - \lfloor x \rfloor}$ учтём, что

$$e^{\lfloor x \rfloor} \leqslant 3^{\lfloor x \rfloor}.$$

Тогда

$$\Delta e^{x - \lfloor x \rfloor} > \frac{10^\kappa}{3^{\lfloor x \rfloor + 1}}$$

и точность величины $e^{x - \lfloor x \rfloor}$ должна составить

$$\kappa - \left\lceil \lg 3^{\lfloor x \rfloor + 1} \right\rceil.$$

Обратимся к случаю $x < -1$. Заметим, что

$$e^x = \frac{1}{e^{-x}},$$

где $-x$ — положительное число. При этом

$$\Delta e^{-x} = e^{-2x} \cdot 10^\kappa > 10^\kappa,$$

то есть точность вычисления e^{-x} должна быть не хуже, чем точность e^x.

1.5.2 Нижняя оценка порядка экспоненты

В том случае, если аргумент экспоненты x по модулю не превышает единицу, нижняя оценка её порядка производится по первому члену ряда Маклорена. В противном случае затем, что

$$\lg e^x = \lg e^{\lfloor x \rfloor} + \lg e^{x - \lfloor x \rfloor},$$

откуда

$$\lfloor \lg e^x \rfloor \geqslant \lfloor \lg e^{\lfloor x \rfloor} \rfloor + \lfloor \lg e^{x - \lfloor x \rfloor} \rfloor.$$

Ранее было замечено, что

$$1 \leqslant e^{x-\lfloor x \rfloor} < e,$$

а значит,

$$0 \leqslant \lg e^{x-\lfloor x \rfloor} < \lg e.$$

По свойству операции округления имеем

$$0 \leqslant \lfloor \lg e^{x-\lfloor x \rfloor} \rfloor \leqslant \lfloor \lg e \rfloor = 0,$$

то есть $\lfloor \lg e^{x-\lfloor x \rfloor} \rfloor = 0$. В этом случае, за нижнюю оценку порядка величины e^x можно принять

$$n_f = \lfloor \lg e^{\lfloor x \rfloor} \rfloor.$$

Однако операция возведения в целую степень числа e сама по себе довольно сложная, а потому для простоты примем $n_f = \lfloor \lg 2^{\lfloor x \rfloor} \rfloor$.

2 Практическая реализация вычислительных алгоритмов

2.1 Реализация инструментов общего назначения

2.1.1 Алгоритмы поиска с предикатами на ограниченном интервале

Поиск количества подлежащих вычислению членов ряда сводится к поиску наименьшего целого положительного числа, которое удовлетворяет определённому предикату. Предикат $P(x)$, который истинен для всех значений $x \geqslant x_0$ и ложен для всех $x < x_0$, будем называть монотонно возрастающим. Если же предикат возвращает истину для любого $x \leqslant x_0$ и ложь для $x > x_0$, то он монотонно убывающий.

Поскольку поиск того или иного значения — задача, обладающая большим множеством решений, удобно создать общий интерфейс. Для случая, когда необходимо найти некоторый элемент в известном диапазоне значений, такой интерфейс имеет следующий вид:

Листинг 2.1. Интерфейс `BoundedFinder`

```
1 @FunctionalInterface
2 public interface BoundedFinder<T extends Comparable<T>>
      {
3     Optional<T> find(T lowerBound, T upperBound);
4 }
```

Метод `find(T, T)` предполагает поиск неизвестного заранее элемента в заданном интервале значений. При этом должны быть каким-то образом заданы правила, в соответствии с которыми осуществляется поиск. Конечно, на множестве обрабатываемых элементов должно быть задано отношение порядка, то есть должна быть возможность сравнивать эти элементы между собой. Для этого достаточно того, чтобы объекты, которые представляют эти элементы, реализовывали интерфейс `Comparable`.

Достаточно эффективной разновидностью поиска в заданном интервале

считается бинарный поиск. Именно на нём основан функционал абстрактного класса `ThresholdFinder`, реализующего `BoundedFinder`. При его создании необходимо задать предикат, на соответствие которому будут исследоваться элементы диапазона, и указать характер предиката: возрастающий или убывающий. При этом предикат должен быть строго монотонным:

Листинг 2.2. Коструктор класса `ThresholdFinder`

```
1 public ThresholdFinder(Predicate<T> predicate, boolean
    increasing) {
2     Objects.requireNonNull(predicate, "Predicate cannot
        be null");
3     this.predicate = predicate;
4     this.increasing = increasing;
5 }
```

Этот класс предназначен для поиска величины x_0, о которой было сказано в начале параграфа.

Главный метод `find(T, T)` класса `ThresholdFinder` сначала проверяет корректность переданных ему границ, а затем осуществляет поиск следующим способом:

Листинг 2.3. Модифицированный бинарный поиск

```
1  T result = null;
2  while (lowerBound.compareTo(upperBound) <= 0) {
3      T midPoint = computeMidPoint(lowerBound, upperBound
         );
4
5      boolean testResult = predicate.test(midPoint);
6      if (testResult) {
7          result = midPoint;
8      }
9
10     if ((testResult && increasing) || (!testResult && !
         increasing)) {
11         upperBound = decrement(midPoint);
12     } else {
```

```
13            lowerBound = increment(midPoint);
14        }
15    }
```

Цикл длится до пересечения верхней и нижней границ поиска. Предполагается реализация метода `computeMidPoint(T, T)`, который возвращает середину диапазона. Если на каком-то из значений предикат возвращает истину, это значение помещается в переменную `result`. Методы `decrement(T)` и `increment(T)` призваны предотвращать зацикливание программы. Впоследствии возвращается значение `Optional.of(result)` или `Optional.empty()`, если значение не было найдено.

Конкретную реализацию методов класса `ThresholdFinder` для целочисленных значений предоставляет класс `IntegerFinder`.

2.1.2 Алгоритмы поиска с предикатами на неограниченном интервале

В случае поиска количества подлежащих вычислению членов ряда речь идёт о неограниченном диапазоне значений $[0; +\infty)$. Для этих целей подходит интерфейс

Листинг 2.4. Интерфейс `SemiBoundedFinder`

```
1 public interface SemiBoundedFinder<T extends Comparable
    <T>> {
2     Optional<T> find(T startingPoint);
3 }
```

Его абстрактная реализация — `AdaptiveThresholdFinder` — предполагает задание предиката, его типа и объекта `ThresholdFinder`. Последнему будет делегирован поиск значения, когда будут определены границы поиска.

Итак, после валидации переменной `startingPoint` выполняются следующие действия:

Листинг 2.5. Интерфейс `SemiBoundedFinder`

```
1 T endingPoint = computeEndingPoint(startingPoint);
2 while (!predicate.test(endingPoint)) {
3     startingPoint = endingPoint;
4     endingPoint = computeEndingPoint(endingPoint);
```

```
5 }
```

Затем задача поиска передаётся объекту `ThresholdFinder`.

Эффективное определение верхней (или нижней в случае убывающего предиката)
границы осуществляется экспоненциальным поиском. На этом основана реализация метода `computeEndingPoint(T)` в классе `AdaptiveIntegerFinder` — наследнике `AdaptiveThresholdFinder`:

Листинг 2.6. Экспоненциальное определение границ

```
1  protected Integer computeEndingPoint(Integer previous)
     {
2      if (endBound == 0) {
3          return increasing ? 1 : -1;
4      }
5
6      try {
7          if (endBound < 0) {
8              return increasing
9                  ? Math.ceilDiv(endBound, 2)
10                 : Math.multiplyExact(endBound, 2);
11         }
12         return increasing
13             ? Math.multiplyExact(endBound, 2)
14             : Math.floorDiv(endBound, 2);
15     } catch (ArithmeticException e) {
16         throw new ArithmeticException("Overflow
             occurred during boundary calculation");
17     }
18 }
```

В том случае, если текущая граница положительна и предикат возрастает, логично расширить границы поиска, умножая их на два. Если же предикат убывает, разумно их уменьшать, деля на два. Для отрицательных границ действуют противоположные правила.

2.1.3 Ленивая инициализация

Многие операции, выполнение которых требуется для приближения значения той или иной функции, достаточно дорогостоящи. Поэтому удобно пользоваться паттерном ленивой инициализации. Для хранения значения, которое требуется инициализировать не раньше, чем оно понадобится, служит класс `ConcurrentLazyHolder`. Он содержит методы, адаптированные для многопоточной среды. Так, базовый метод `getResource()` защищён от состояния гонки:

Листинг 2.7. Получение значения

```
1  public T getResource() {
2      if (resource == null) {
3          synchronized (this) {
4              if (resource == null) {
5                  try {
6                      resource = initializer.get();
7                  } catch (Exception e) {
8                      throw new IllegalStateException("
                           Failed to initialize resource", e
                           );
9                  }
10             }
11         }
12     }
13
14     return resource;
15 }
```

Прочие методы реализованы очевидным образом.

2.1.4 Интервалы и валидация аргумента

Для работы с интервалами реализован интерфейс `Interval`. Он предоставляет методы, позволяющие проверить, принадлежит ли число заданному интервалу или находится справа или слева от него. Для взаимодействия с интервалами достаточно использовать фабрики `IntervalFactory` (позволяет со-

здавать открытые, закрытые и прочие распространённые типы интервалов) и `CommonIntervalFactory` (создаёт интервалы из положительных чисел, отрицательных чисел и другие специализированные интервалы).

Для валидации аргумента создан функциональный интерфейс `Validator`. Создание его экземпляров делегируется фабрикам `IntervalValidatorFactory` и `CommonValidatorFactory`. Их функционал аналогичен фабрикам интервалов.

2.2 Числовые типы и операции с порядком и точностью чисел

2.2.1 Десятичные числа произвольной точности

Десятичные числа произвольной точности — это тип чисел, наиболее пригодный для точных вычислений. В Java они представлены классом `BigDecimal` . Конечно, фактически количество значащих цифр, которые позволяет хранить объект этого класса, ограничено. На практике ограничение вносит доступная в системе память. В теории же количество цифр может достигать величины $2^{31} - 1$, то есть максимального значения, которое может хранить переменная типа `int`. Это более чем приемлемо для большинства точных вычислений.

Одним из основных свойств объектов класса `BigDecimal` является масштаб (`scale`). В соответствии с документацией, масштаб, если он неотрицателен, равен количеству цифр в числе после десятичного разделителя. Если же масштаб меньше нуля, то его модуль есть количество нулей, которые должны быть дописаны справа к числу в его несмещённом представлении. Несмещённое значение числа (unscaled value) — это целое число без учёта десятичной точки и масштаба. Таким образом, имеет смысл равенство:

$$k = -\text{scale},$$

где k — индекс наименьшего учтённого разряда. Ранее вводимое ограничение $\text{scale} \geqslant 0$здесь опускается. В таком случае значение числа определяется как

$$\text{unscaledValue} \cdot 10^{-\text{scale}}.$$

Одно и то же число может быть представлено в виде разных пар несме-

щённого значения и масштаба. Метод `equals(BigDecimal)`, реализованный в классе `BigDecimal`, сравнивает именно эти поля объектов. И даже если числа математически совпадают, но имеют разные представления, метод вернёт `false`. Однако метод `compareTo(BigDecimal)` сравнивает числа математически и потому более предпочтителен во многих случаях.

Другое интересующее нас свойство `BigDecimal` — точность (`precision`). Точность объекта `BigDecimal` — это общее количество значащих цифр в числе (ранее мы обозначали эту величину через λ). При этом, если запись числа содержит нули в младших разрядах, они также учитываются как в точности, так и в масштабе. Иными словами, можно сказать, что точность — это длина, то есть количество цифр в несмещённом значении числа.

2.2.2 Целые числа произвольной точности

Целые числа произвольной точности представлены классом `BigInteger`. Мы будем пользоваться этим классом в основном, чтобы избежать проверки того или иного значения на принадлежность к целым числам. В противном случае удобнее использовать класс `BigDecimal`.

Метода, позволяющего определить количество цифр в числе `BigInteger`, не существует, но это несложно сделать косвенным образом. Например, можно перевести абсолютное значение числа в строку и затем вычислить её длину. Если число отрицательное, его строковое представление будет иметь знак минус в самом начале, поэтому следует переводить в строку именно модуль числа. Реализация соответствующего метода представлена в листинге 2.8.

Листинг 2.8. Метод для определения точности числа `BigInteger`

```
1 public static int precision(BigInteger number) {
2     String stringAbsValue = number.abs().toString();
3     return stringAbsValue.length();
4 }
```

Любое целое число имеет единственное представление в формате `BigInteger`, поэтому допустимо использование метода `equals(BigInteger)` при проверке чисел на равенство.

2.2.3 Примитивные числовые типы

Примитивные числовые типы с меньшим диапазоном значений могут быть преобразованы в типы с большим диапазоном без потери данных благодаря механизму автоматического приведения типов (implicit casting). Этот процесс называется расширяющим преобразованием (widening conversion). Преобразования возможны по следующим схемам:

$$\texttt{byte} \to \texttt{short} \to \texttt{int} \to \texttt{long} \to \texttt{float} \to \texttt{double},$$
$$\texttt{char} \to \texttt{int} \to \texttt{long} \to \texttt{float} \to \texttt{double}.$$

Во многих случаях, когда необходим метод, обрабатывающий произвольное число, удобно реализовать его для чисел типа `double`, поскольку к нему могут быть приведены все прочие примитивы. Также этот метод позволит работать с классами-обёртками, поскольку они приводятся к соответствующим примитивам путём автоматической упаковки и распаковки (autoboxing и unboxing).

2.2.4 Вычисление порядка числа

Масштаб и точность объекта `BigDecimal` позволяют однозначно определить его порядок n по формуле:

$$n = k + \lambda - 1 = \text{precision} - \text{scale} - 1.$$

Для подсчёта порядка такого числа реализован статический метод `OrderUtil.orderOf(BigDecimal)`, представленный в листинге 2.9. В случае передачи числа 0 в качестве аргумента, он возвращает `Integer.MIN_VALUE`, несмотря на то, что порядок для этого числа на самом деле не определён. Это сделано, чтобы избежать выброса неожиданных исключений, исходя из следующего соображения:

$$\lim_{x \to 0} \lg x = -\infty.$$

Листинг 2.9. Метод поска порядка числа `BigDecimal`

```
1 public static int orderOf(BigDecimal number) {
2     if (number.compareTo(BigDecimal.ZERO) == 0) {
3         return Integer.MIN_VALUE;
4     }
```

```
5
6       return number.precision() - number.scale() - 1;
7   }
```

Для целых чисел формата `BigInteger` имеем $k = 0$. Тогда, по формуле (1.4)

$$n = \lambda - 1 = \text{precision} - 1.$$

Реализация алгоритма поиска порядка таких чисел представлена в листинге 2.10.

Листинг 2.10. Метод поска порядка числа `BigInteger`

```
1 public static int orderOf(BigInteger number) {
2       if (number.equals(BigInteger.ZERO)) {
3           return Integer.MIN_VALUE;
4       }
5
6       return precision(number) - 1;
7   }
```

Для чисел, представимых в формате примитивных типов, возможно определение порядка путём вычисления целого значения десятичного логарифма с помощью метода `Math.log10(double)`. Однако этот метод до округления вернёт значение логарифма с точностью, которую позволяет хранить переменная типа `double`, что для нас избыточно. В таком случае проще преобразовать число в объект `BigDecimal` и затем воспользоваться методом `Calculator.order(BigDecimal)`. Перегрузка этого метода для типа `double` представлена в листинге 2.11.

Листинг 2.11. Метод поска порядка числа приммитивного типа|

```
1 public static int order(double number) {
2       BigDecimal bigDecimalValue = BigDecimal.valueOf(
          number);
3       return order(bigDecimalValue);
4   }
```

Важно отметить, что следует использовать метод `BigDecimal.valueOf(double)`, а не конструктор `BigDecimal(double)`. Это связано с тем, что кон-

структор преобразует число типа `double` непосредственно в `BigDecimal`, что может привести к проблемам, поскольку числа типа `double` могут неточно представлять десятичные значения из-за особенностей двоичного представления чисел с плавающей точкой. Метод `BigDecimal.valueOf(double)` решает эту проблему, используя точное строковое представление числа.

2.2.5 Вычисление верхней оценки порядка числа

Как было выяснено ранее, нижняя оценка порядка совпадает с самим порядком n, поэтому введение отдельных методов для её вычисления не требуется. Верхняя оценка порядка числа равна порядку n только в том случае, если его мантисса равна ± 1. В этом случае число можно записать в виде

$$10^n = 1\underbrace{0\ldots0}_{n}, \tag{2.1}$$

если $n \geqslant 0$, и

$$10^n = 0,\underbrace{0\ldots0}_{n-1}1,$$

если $n < 0$.

Для вычисления верхней оценки порядка числа a формата `BigDecimal` сначала вычислим его порядок n. Затем сдвинем десятичную точку влево на n, если $n \geqslant 0$, или вправо на $-n$ позиций в обратном случае. Эта операция эквивалентна умножению числа на 10^{-n}:

$$a \cdot 10^{-n} = \alpha \cdot 10^n \cdot 10^{-n} = \alpha.$$

Таким образом, мы получим его мантиссу α. Если она по модулю равна единице, то верхняя оценка порядка совпадает с порядком числа a. В противном случае верхняя оценка порядка равна $n+1$. Реализация представлена в листинге 2.12

Листинг 2.12. Метод поска верхней оценки порядка числа `BigDecimal`

```
1 public static int overestimateOrderOf(BigDecimal number
    ) {
2     int order = orderOf(number);
3     BigDecimal significand = number.movePointLeft(order
        );
```

```
4
5       if (significand.abs().compareTo(BigDecimal.ONE) ==
            0) {
6           return order;
7       }
8
9       return order + 1;
10  }
```

В случае с целым числом `BigInteger`, возможно только его представление вида (2.1). Значит, можно ограничиться проверкой строкового представления числа на соответствие регулярному выражению “`-?10*`”, Это соответствует числу, содержащему единицу в старшем разряде и нули в остальных. Реализация представлена в листинге 2.13.

Листинг 2.13. Метод поска верхней оценки порядка числа `BigInteger`

```
1  public static int overestimateOrderOf(BigInteger number
       ) {
2       String stringAbsValue = number.abs().toString();
3       int order = orderOf(number);
4
5       if (stringAbsValue.matches("-?10*")) {
6           return order;
7       }
8
9       return order + 1;
10  }
```

Если нужно оценить порядок числа примитивного типа, преобразуем его в объект `BigDecimal`, а затем воспользуемся уже готовым методом.

2.2.6 Вычисление индекса младшего разряда числа

В дальнейшем средства, вычисляющие приближённое значение числа, будут оперировать индексом младшего разряда, значение которого необходимо вычислить. Для удобства использования этих инструментов реализуем методы перевода масштаба и точности в значение индекса младшего разряда.

Методы для вычисления индекса младшего разряда представлены в листинге 2.14.

Листинг 2.14. Методы поиска индекса младшего разряда числа

```
1  public static int getLeastDigitPositionByScale(int
     scale) {
2      return -scale;
3  }
4
5  public static int getLeastDigitPositionByPrecision(int
     precision, int order) {
6      if (precision < 1) {
7          throw new ArithmeticException("Precision cannot
               be less than one");
8      }
9
10     return order + 1 - precision;
11 }
```

2.3 Реализация и обработка числовых рядов

2.3.1 Обработка отдельных членов ряда

Для обработки членов ряда по отдельности создан параметризованный интерфейс `SeriesTerm`. Он предоставляет следующие методы:

- `approximate(int termIndex, int accuracy)` — для приближения значения члена с заданным индексом;
- `approximateMinimal(int termIndex)` — для приближения значения члена с минимальной возможной точностью (до первой значащей цифры; может быть использовано для вычисления порядка);
- `overestimateOrder(int termIndex)` — для получения верхней оценки порядка члена.

От этого интерфейса непосредственно наследуется класс `BigSignedTerm`, который позволяет управлять множителем вида $(-1)^n$. При этом предполагается

реализация методов `approximateUnsigned(int termIndex, int accuracy)` и `approximateUnsignedMinimal(int termIndex)`, которые затем используются для применения знака. Следующим в цепочке наследования идёт класс `BigFractionalTerm`, созданный для представления членов, состоящих из числителя и знаменателя и упрощения работы с ними. Конкретные реализации членов степенных рядов используют базовые возможности языка Java.

2.3.2 Накопление членов ряда

Для накопления членов ряда создан интерфейс `SeriesAccumulator`. Он предоставляет возможность последовательно суммировать члены ряда заданного типа. Его конкретная реализация — `BigSeriesAccumulator` — адаптирована к использованию в многопоточной среде. Сначала определяется значение индекса последнего необходимого члена ряда по начальному индексу суммирования и общему количеству членов. Затем вызывается метод `executeComputations` который запускает вычисления в индивидуальных потоках:

Листинг 2.15. метод `executeComputations`

```
1 private CompletionService<BigDecimal>
    executeComputations(int minSeriesIndex, int
    greatestIndex, int termScale, SeriesTerm<BigDecimal>
    seriesTerm) {
2     int availableProcessors = Runtime.getRuntime().
        availableProcessors();
3     ExecutorService executor = Executors.
        newFixedThreadPool(availableProcessors);
4     CompletionService<BigDecimal> completionService =
        new ExecutorCompletionService<>(executor);
5
6     for (int i = minSeriesIndex; i <= greatestIndex; i
        ++) {
7         final int termIndex = i;
8         Callable<BigDecimal> termCallable = () ->
            seriesTerm.approximate(termIndex, termScale);
9         completionService.submit(termCallable);
```

```
10      }
11
12      executor.shutdown();
13      return completionService;
14  }
```

При этом количество потоков вычисляется исходя из количества доступных процессоров, чтобы не перегружать систему.

Далее, посредством взаимодействия с объектом `CompletionService<BigDecimal>`, который возвращает метод `executeComputations` производится само суммирование:

Листинг 2.16. метод `computeSum`

```
1  private static BigDecimal computeSum(int termsNumber,
     CompletionService<BigDecimal> completionService) {
2       BigDecimal sum = BigDecimal.ZERO;
3
4       for (int i = 0; i < termsNumber; i++) {
5           try {
6               Future<BigDecimal> futureAugend =
                    completionService.take();
7               BigDecimal augend = futureAugend.get();
8               sum = sum.add(augend);
9           } catch (InterruptedException |
                ExecutionException e) {
10              throw new RuntimeException("Caught
                    exception while accumulating terms", e);
11          }
12      }
13      return sum;
14  }
```

2.3.3 Объектно-ориентированное представление рядов

Для представления ряда создан абстрактный класс `Series`. Он обладает обширным набором параметров, например:

- `boolean isOptimized` — флаг, положительное значение которого означает, что члены ряда удовлетворяют соотношению $|R_{n+1}| \leqslant \frac{1}{10}|R_n|$;
- `int minSeriesIndex` — начальный индекс суммирования;
- `SeriesTerm<T> seriesTerm` — объект, описывающий член ряда;
- `SeriesAccumulator<T> seriesAccumulator` — утилита накопления суммы.

Этот класс реализует методы, общие для всех рядов:

- `computeTermAccuracy` — вычисляет погрешность, с которой должны быть приближены отдельные члены;
- `accumulateSum` — накапливает необходимую сумму ряда;
- `requiredTermsCount` — вычисляет количество членов ряда, которые должны быть учтены.

Эти методы реализованы в соответсвии с полученными ранее теоретическими выводами.

Класс `BigSeries` — прямой наследник класса `Series` — предоставляет возможность для всех рядов единообразно получать утилиту, вычисляющую порядок их бесконечной суммы. Для генерации конкретных рядов создан интерфейс `BigSeriesFactory`.

2.4 Реализация приближений элементарных функций

Все инструменты для вычисления приближённых значений наследуют интерфейс `Approximator`. Процесс создания конкретных реализаций этого интерфейса может быть трудоёмким и неинтуитивным, поэтому была создана фабрика `ApproximationUtil`, которая позволяет легко приближать значения таких функций, как логарифм и экспонента. Эти инструменты работают по определённым ранее правилам.

Заключение

В ходе выполнения данной работы были объединены теоретические знания и практические навыки для разработки эффективных методов приближённых вычислений элементарных функций. Рассмотренные численные методы и их реализация на языке Java позволили создать инструменты, обеспечивающие высокую точность и производительность вычислений для функций, таких как экспонента и логарифм.

Использование числовых типов произвольной точности, таких как `BigDecimal` и `BigInteger`, позволило преодолеть ограничения стандартных примитивных типов и избежать потери значащих цифр при арифметических операциях. Это особенно важно при работе с бесконечными рядами и требует тщательного управления порядком и точностью чисел.

Разработанная архитектура с использованием интерфейсов `SeriesTerm`, `SeriesAccumulator` и `Approximator`, а также их конкретных реализаций, обеспечила модульность и гибкость системы. Применение объектно-ориентированного подхода и паттернов проектирования упростило расширение функциональности и адаптацию к различным задачам.

Эффективность вычислений была достигнута за счёт оптимизации алгоритмов и использования многопоточности, что позволило существенно сократить время расчётов при работе с большими объёмами данных. Это делает разработанные инструменты пригодными для применения в областях, где требуется высокая точность и скорость вычислений, таких как научные исследования, инженерные расчёты и финансовый анализ.

В перспективе возможны дальнейшие улучшения и расширения проекта. Это включает реализацию приближений для дополнительных элементарных функций, исследование альтернативных численных методов, а также интеграцию с другими системами и приложениями. Дополнительное внимание может быть уделено оптимизации производительности и управлению ресурсами в условиях высоконагруженных вычислений.

Таким образом, поставленные цели были успешно достигнуты. Работа продемонстрировала, как теоретические основы численных методов могут быть эффективно применены на практике с использованием современных технологий программирования. Полученные результаты вносят вклад в развитие методов приближённых вычислений и открывают возможности для дальнейших

исследований в этой области.

Список литературы

[1] Г. Е. Иванов. *Лекции по математическому анализу: Учебное пособие. В двух частях. Часть 1*. 4-е, переработанное и дополненное. Москва: МФТИ, 2019. ISBN: 978-5-7417-0713-5.

[2] Дональд Эрвин Кнут. *Искусство программирования, том 3. Сортировка и поиск*. 2-е, научно-популярное. Москва: ИД Вильямс, 2018. ISBN: 978-5-8459-0082-1.

[3] А. М. Тер-Крикорова и М. И. Шабунин. *Курс математического анализа: Учебное пособие для вузов*. 8-е, электронное. Москва: Лаборатория знаний, 2020. ISBN: 978-5-00101-702-8.

[4] *Численные методы: Линейная алгебра и нелинейные уравнения*. Москва: Высшая школа, 2000. ISBN: 5-06-003654-5.